

EV355227254

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

**SYSTEM AND METHOD FOR PERFORMING
PATH-SENSITIVE VALUE FLOW ANALYSIS
ON A PROGRAM**

Inventors:

Manuvir Das

Stephen Adams

Nurit Dor

Seth Hallem

ATTORNEY'S DOCKET NO. MS1-1734US

SYSTEM AND METHOD FOR PERFORMING PATH-SENSITIVE VALUE FLOW ANALYSIS ON A PROGRAM

TECHNICAL FIELD

The subject matter relates generally to analysis of software programs and, in particular, to performing flow analyses.

BACKGROUND

Today, some software programs contain one million lines of code (LOC) or more. In order to ensure the reliability of these software programs, the software programs undergo testing before releasing the software programs to consumers. Typically, the testing includes a value flow analysis. Value flow analysis analyzes the software program to determine which memory locations hold a given value at a given program point along a given execution path within the program. The analysis then identifies code that incorrectly uses a value. For example, the analysis may check whether a function call has acquired a lock that was created by a preceding function call, whether a value is valid for a given function call, and the like. Because values created in one portion of the code may be passed to numerous other portions of the code, value flow analysis tracks each execution path for every value.

The current approaches for value flow analysis make a trade-off between precision and scalability. If the value flow analysis is precise, the analysis maintains information about all the values for each execution path. When the

1 software program is very large, this precise value flow analysis can not compute
2 the necessary information in a timely manner. Thus, precise value flow analysis
3 does not scale well to large software programs. In contrast, imprecise value flow
4 analysis does scale well to large software programs. However, imprecise value
5 flow analysis does not keep accurate information. Rather, at certain locations
6 within the program, the information is merged. Because, the imprecise value flow
7 analysis merges some of the information, the results identify some portions of the
8 code as having errors, when in fact those portions do not have errors. This
9 reporting of incorrect errors is commonly referred to as noise. If the imprecise
10 value flow analysis has too much noise, the analysis is not useful. Thus, full-scale
11 reliable value flow analysis of a software program having a large code base has
12 been unattainable.

14 SUMMARY

15 A method and system for performing path-sensitive value flow analysis on
16 a software program is provided. Concrete state and value alias information is
17 tracked along each statement and each relevant path in an abstract program and is
18 stored as a symbolic state in a symbolic store. The value alias information
19 includes a first set of aliases that identify aliases for a designated value that is
20 being analyzed and a second set of aliases that identify possible aliases for the
21 designated value. The value alias information is obtained using imprecise memory
22 alias analysis. Along each relevant path for each statement, transforms are applied
23 to the sets of aliases to update the first and second sets of aliases. The transforms
24 are applied based on the type of statement being processed. Symbolic states
25

existing at the same location are merged if the value alias information is identical within the symbolic states.

BRIEF DESCRIPTION OF THE DRAWINGS

FIGURE 1 illustrates an exemplary computing device that may be used in one exemplary embodiment of the path-sensitive value flow analysis.

FIGURE 2 is a block diagram illustrating an exemplary environment for practicing the path-sensitive value flow analysis.

FIGURE 3 is a logical flow diagram illustrating an overview of a path-sensitive value flow analysis process.

FIGURE 4 is a logical flow diagram illustrating a process for tracking the concrete state and value alias information in the symbolic store for use in the path-sensitive value flow analysis process shown in FIGURE 3.

FIGURE 5 is a logical flow diagram illustrating a process for adding symbolic states to the symbolic store for use in the tracking process illustrated in FIGURE 4.

FIGURE 6 is a table illustrating transforms for updating a must set of the value alias information for use in FIGURE 4.

FIGURE 7 is a table illustrating transforms for updating a may set of the value alias information for use in FIGURE 4.

FIGURES 8-10 are a series of control flow graphs that represent the logic of an exemplary software program and illustrate the symbolic states at various

1 points along the control flow graph during the path-sensitive value flow analysis
2 process.

4 DETAILED DESCRIPTION

5 Briefly stated, the present path-sensitive value flow analysis tracks the flow
6 of values within source code written for a software program. The path-sensitive
7 value flow analysis process maintains a concrete state of the program and value
8 alias information for the program. The value alias information identifies a set of
9 variables which are associated with a value being analyzed. The set of variables
10 may be in one of two sets. A first set identifies the variables that "must" be
11 associated with the analyzed value. The second set identifies the variables that
12 "may" be associated with the analyzed value. The first and second sets are
13 determined by performing transform functions based on the type of statement that
14 is being processed in the software program.

15 Using the concrete state, the analysis determines which paths of a branch
16 statement are relevant paths. Upon reaching a join point associated with the
17 traversed relevant paths, the concrete states resulting from each relevant path are
18 combined based on a comparison of the value alias information. The
19 determination of the value alias information and the combining of the value alias
20 information allow the present path-sensitive value flow analysis to scale to
21 software programs with a large code base without experiencing exponential
22 growth in the search space. Thus, the value flow analysis process provides
23 accurate results for software programs having any size code base (e.g., a million
24 LOCs).

1 The following description sets forth a specific embodiment of a path-
2 sensitive value flow analysis process that incorporates elements recited in the
3 appended claims. The embodiment is described with specificity in order to meet
4 statutory requirements. However, the description itself is not intended to limit the
5 scope of this patent. Rather, the inventors have contemplated that the claimed
6 invention might also be embodied in other ways, to include different elements or
7 combinations of elements similar to the ones described in this document, in
8 conjunction with other present or future technologies.

9 Illustrative Operating Environment

10 With reference to FIGURE 1, one exemplary system for implementing the
11 path-sensitive value flow analysis includes a computing device, such as computing
12 device 100. In a very basic configuration, computing device 100 typically
13 includes at least one processing unit 102 and system memory 104. Depending on
14 the exact configuration and type of computing device, system memory 104 may be
15 volatile (such as RAM), non-volatile (such as ROM, flash memory, etc.) or some
16 combination of the two. System memory 104 typically includes an operating
17 system 105, one or more program modules 106, and may include program data
18 107. This basic configuration is illustrated in FIGURE 1 by those components
19 within dashed line 108.

20 Computing device 100 may have additional features or functionality. For
21 example, computing device 100 may also include additional data storage devices
22 (removable and/or non-removable) such as, for example, magnetic disks, optical
23 disks, or tape. Such additional storage is illustrated in FIGURE 1 by removable
24 storage 109 and non-removable storage 110. Computer storage media may
25

1 include volatile and nonvolatile, removable and non-removable media
2 implemented in any method or technology for storage of information, such as
3 computer readable instructions, data structures, program modules, or other data.
4 System memory 104, removable storage 109 and non-removable storage 110 are
5 all examples of computer storage media. Computer storage media includes, but is
6 not limited to, RAM, ROM, EEPROM, flash memory or other memory
7 technology, CD-ROM, digital versatile disks (DVD) or other optical storage,
8 magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage
9 devices, or any other medium which can be used to store the desired information
10 and which can be accessed by computing device 100. Any such computer storage
11 media may be part of device 100. Computing device 100 may also have input
12 device(s) 112 such as keyboard, mouse, pen, voice input device, touch input
13 device, etc. Output device(s) 114 such as a display, speakers, printer, etc. may
14 also be included. These devices are well known in the art and need not be
15 discussed at length here.

16 Computing device 100 may also contain communication connections 116
17 that allow the device to communicate with other computing devices 118, such as
18 over a network. Communication connections 116 is one example of
19 communication media. Communication media may typically be embodied by
20 computer readable instructions, data structures, program modules, or other data in
21 a modulated data signal, such as a carrier wave or other transport mechanism, and
22 includes any information delivery media. The term "modulated data signal"
23 means a signal that has one or more of its characteristics set or changed in such a
24 manner as to encode information in the signal. By way of example, and not
25

1 limitation, communication media includes wired media such as a wired network or
2 direct-wired connection, and wireless media such as acoustic, RF, infrared and
3 other wireless media. The term computer readable media as used herein includes
4 both storage media and communication media.

5 Value Flow Analysis Environment

6
7 FIGURE 2 is a block diagram illustrating an exemplary environment for
8 practicing the present path-sensitive value flow analysis. The exemplary
9 environment shown in FIGURE 2 is a value flow analysis environment **200**. The
10 goal in this value flow analysis environment **200** is to verify that the values
11 assigned to variables (e.g., variable **203**) within program **204** (i.e., source code)
12 will operate properly in any reachable state. The value flow analysis environment
13 **200** includes a value flow analysis tool **202**. The value flow analysis tool **202** may
14 operate as a stand-alone tool (i.e., a program module **106** in FIGURE 1) or may
15 operate within another software verification tool, such as a compiler application
16 (not shown). The results from the value flow analysis tool may help identify
17 errors within the software program, help stream-line a compile process, and the
18 like. Program **204** may be written using any programming language, such as "C"
19 programming language and the like.

20 In this value flow analysis environment **200**, the value flow analysis tool
21 **202** receives the program **204**. The value flow analysis tool **202** or another tool
22 translates the program **204** into an abstract program **205**. The value flow analysis
23 tool **202** then performs the path-sensitive value flow analysis process on the
24 abstract program **205**.
25

1 During the path-sensitive value flow analysis process, the value flow
2 analysis tool **202** maintains a symbolic store **210** and may maintain a worklist **230**.
3 The symbolic store **210** and the worklist **230** may reside in RAM, on a hard disk,
4 or on any other type of storage. The symbolic store **210** may include several
5 symbolic states (e.g., symbolic states **212-216**). Each symbolic state includes
6 value alias information (e.g., **220**). In addition, each symbolic state may include a
7 concrete state (e.g., concrete state **222**) associated with the value alias information.
8 In general, the value alias information **220** identifies the variables in which the
9 value being analyzed may currently be stored and the concrete state **222** identifies
10 information on other properties, such as values of other variables. Each symbolic
11 state is associated with some edge in a control flow graph.

12 Similarly, the worklist **230** may include several symbolic states (e.g.,
13 symbolic states **232 - 236**) with each having value alias information (e.g., **238**) and
14 a concrete state (e.g. **240**). Typically, the symbolic states **232 - 236** within the
15 worklist **230** are a subset of the symbolic states **212-216** in the symbolic store **210**.
16 As will be described in detail below, the value flow analysis tool **202** uses the
17 worklist **230** for determining which symbolic states still need to be processed. On
18 the other hand, the symbolic store **210** is used to determine whether a newly
19 created symbolic state must be merged with previously created states. The
20 symbolic store is also used to determine whether any of the symbolic states
21 indicate an error condition, once processing of the abstract program is complete.

22 The value flow analysis tool **202** traverses each relevant path in the abstract
23 program **205**. When any statement in the abstract program **205** affects the value
24 being analyzed, the value flow analysis tool **202** updates the symbolic store **210** to
25

1 accurately reflect the changes. The value flow analysis tool 202 outputs results
2 224 from processing. The results 224 may be incrementally output, outputted
3 upon completion of the process, or the like. Briefly, as will be described in detail
4 later in conjunction with FIGURES 3-5 and the related series of control flow
5 graphs (CFGs) shown in FIGURES 8-10, the value flow analysis tool 202
6 performs concepts derived from dataflow analysis and includes a unique tracking
7 of value alias information and an add heuristic that provides accurate results 224
8 without experiencing exponential search space explosion.

9 Value Flow Analysis Process

10 FIGURE 3 is a logical flow diagram illustrating an overview of a path-
11 sensitive value flow analysis process. The process enters at starting block 301,
12 where the value flow analysis tool has been initiated in some manner (e.g., a
13 variable is instantiated). The process 300 continues at block 302.

14 At block 302, an abstract program is created. In general, the abstract
15 program is created by converting each statement in the software program 204
16 (FIGURE 2) into one or more of the types of statements listed in the first column
17 of Tables 600 and 700, illustrated in FIGURES 6 and 7, respectively. Therefore,
18 one complex statement in the software program may become several statements in
19 the abstract program. This conversion may be performed using various standard
20 procedures, commonly referred to as "introducing temporaries" or "converting to
21 three address form." Because those skilled in the art know of these various
22 standard procedures, the procedures will not be discussed at length here. In
23 summary, at block 302, the abstract program is created using standard procedures
24 for converting complex statements within the software program into one or more
25

1 simple statements. Once the abstract program is created, processing continues at
2 block 304.

3 At block 304, a symbolic store is created and initialized. Typically,
4 creating the symbolic store includes allocating memory and identifying a type of
5 variable or structure. The symbolic store may take any number of forms. One
6 form is a True-False lattice. In the True-False lattice, elements in the symbolic
7 store are maps from variables V in the program to "true" or "false". Another
8 illustrative form is a constant propagation lattice. In the constant propagation
9 lattice, elements in the symbolic store are maps from variables V in the program to
10 "true" or "false", integers, or floating-point values. Those skilled in the art will
11 appreciate that other forms for the symbolic store may be implemented without
12 departing from the scope of the claimed invention. Processing continues at block
13 306.

14 At block 306, each relevant path in the abstract program is traversed.
15 During traversal of the relevant path, if the concrete state or the value alias
16 information changes, the symbolic store may be updated to reflect the change. As
17 will be described in detail below in conjunction with FIGURE 4, the tracking
18 process uses the symbolic store to determine whether a branch is relevant. As will
19 be described, even when a branch affects the concrete state of the program, the
20 statement may not be relevant to the value being analyzed. Therefore, the present
21 path-sensitive value flow analysis does not traverse non-relevant paths and refrains
22 from unnecessarily doubling the search space upon each branch statement.
23 Processing continues at block 308.
24
25

1 At block 308, results from the value flow analysis process are output. The
2 output may be in any form, such as a visual representation on a display, a file, or
3 the like. A person responsible for testing the software program may then perform
4 further analysis on the value being analyzed and correct any errors that are
5 reported for the program. Processing then ends.

6 FIGURE 4 is a logical flow diagram illustrating a process for tracking the
7 concrete state and the value alias information for the symbolic store for use in the
8 path-sensitive value flow analysis process shown in FIGURE 3. First, the flow of
9 process 400 will be described, along with the flow of process 500 shown in
10 FIGURE 5. Next, the updating of the symbolic state will be described in
11 conjunction with the tables in FIGURES 6 and 7. Finally, an example will be
12 described in conjunction with logical flow diagrams 400 and 500, along with the
13 control flow graphs illustrated in FIGURES 8-10.

14 Before describing process 400, terminology that is used throughout the
15 following discussion is defined. A symbolic store is a collection of symbolic
16 states that are encountered while processing the abstract program. As will be
17 described in detail below, the present path-sensitive value flow analysis reduces
18 the number of symbolic states that are stored in the symbolic store by applying a
19 heuristic that first attempts to combine symbolic states before adding the symbolic
20 states to the symbolic store. A worklist is a current working list of the symbolic
21 states that still need processing. Memory aliasing is having two different names
22 refer to the same location in memory. Memory aliasing occurs when pointers are
23 used.

1 The process enters at starting block **401**, after an abstract program has been
2 created and an empty symbolic store and an empty worklist have been created.
3 For convenience, the following discussion assumes that process **400** is an
4 intraprocedural process having one function (e.g., `main()`) in the program and
5 does not have any function calls or returns. The value flow analysis tool enters the
6 `main()` program. At this time, the symbolic store contains a single symbolic state
7 that includes the value alias information associated with the value being analyzed.
8 The inventors of the present path-sensitive value flow analysis discovered that
9 distributive dataflow problems can be bit-vectorized. In addition, the inventors
10 discovered that value flow analysis is distributive even in the presence of memory
11 aliasing. Thus, process **400** analyzes one value of interest at a time, which reduces
12 the memory footprint needed for value flow analysis. At this point, the worklist
13 also contains the initial symbolic state (described below). The process **400**
14 continues at decision block **402**.

15 At decision block **402**, a determination is made whether there is another
16 symbolic state in the worklist. If there are no more symbolic states in the worklist,
17 processing is complete and proceeds to a return block that proceeds back to
18 FIGURE 3. When processing is complete, if the symbolic store does not contain
19 any symbolic state with an error, the results indicate that the value being analyzed
20 was not used in any incorrect manner throughout the program. Thus, the program
21 is ready for the next phase of testing. However, as long as there is another
22 symbolic state in the worklist, processing continues at block **404**.

23 At block **404**, a symbolic state *S* is retrieved from the worklist. When the
24 abstract program is first entered, typically, the worklist contains one symbolic
25

1 state that specifies the default state. The default state lists one value alias (e.g.,
2 variable). The value assigned to this value alias is the value being analyzed. As
3 will be described in detail below, the value may be assigned to other variables
4 during various execution paths within the program. These other variables will
5 then be added to the symbolic state. Thus, after processing several branch
6 statements, the worklist may contain several symbolic states. Processing
7 continues at block 406.

8 At block 406, the symbolic state S is removed from the worklist, because
9 symbolic state S has been selected for processing. Thus, the worklist maintains a
10 list of interim symbolic states that have not yet been selected for processing.
11 Processing continues at block 408.

12 At block 408, a statement N is retrieved. The statement N is the statement
13 that follows the edge associated with symbolic state S. For the following
14 discussion, a statement is a node in the CFG. Nodes are produced for assignment
15 statements, branch conditions, function calls, join points, return statements, and
16 the like. The nodes are produced in a standard manner that is well known in the
17 art, such as by breaking compound statements in the original program into
18 individual statements in the abstract program. Processing continues to decision
19 block 410.

20 At decision block 410, a determination is made whether the statement N is
21 a branch statement. If statement N is not a branch statement, processing continues
22 to decision block 412. At decision block 412, a determination is made whether
23 statement N affects the value alias information or the concrete state of the
24 symbolic state being processed. As mentioned above, the value alias information
25

1 represents the memory locations in which the value being analyzed may be stored.
2 The concrete state is associated with other aspects of the computation performed
3 by the program, such as the values of other program variables. If the statement
4 affects the symbolic state, processing continues to block 413.

5 At block 413, symbolic state S is updated. If the statement N affected the
6 concrete state, the concrete state S is updated with the new information. For
7 example, for the statement "x=0", block 413 will add "x=0" to the concrete state.
8 The manner in which the concrete state is updated in the symbolic store is well
9 known in the art and is not discussed at length here.

10 On the other hand, if the statement N affects the value alias information, the
11 value alias information is updated in accordance with the present method. In
12 general, the value alias information includes two sets of information. The first set
13 (hereinafter also referred to as the Must set) represents the aliases in which the
14 value being analyzed is stored. The second set (hereinafter also referred to as the
15 May set) represents the aliases in which the value being analyzed may be stored.
16 The union of the Must and the May set is over inclusive and represents all the
17 locations in which the value may be stored. As will be described in detail in
18 conjunction with FIGURES 6 and 7, the determination of the Must and May sets
19 depends on the type of statement and the variables affected by the statement.
20 Once the Must and May sets are updated and the symbolic state S is updated,
21 processing continues at block 414.

22 At block 414, a new symbolic state (i.e., symbolic state S'') is created that
23 contains symbolic state S with updated information. From block 414, processing
24 continues to block 418.
25

1 Referring back to decision block 412, if the statement N does not affect the
2 symbolic state S, processing proceeds to block 416. At block 416, a copy of
3 symbolic state S is created (i.e., symbolic state S"). Processing then also
4 continues to block 418.

5 At block 418, the add heuristic of the present method is applied to the
6 symbolic state S" on the outgoing edge from statement N. The add heuristic (i.e.,
7 add process 500), described in detail below, is one of the reasons that the present
8 path-sensitive value flow analysis achieves accurate analysis and achieves this
9 accurate analysis even for large software programs, not just for small software
10 programs. In general, the add process utilizes the value alias information and the
11 concrete state to determine which symbolic states should be merged and to
12 determine which symbolic states are copied into the symbolic store. The add
13 process is described in detail below in conjunction with FIGURE 5. Processing
14 then loops back to decision block 402 for another symbolic state, if available.

15 Now, returning to decision block 410, if statement N is a branch statement,
16 processing continues to block 420. At block 420, copies of the symbolic state
17 (e.g., copies S' and S") are created for each path that may be traversed. Each path
18 then updates its copy of the symbolic state as needed. Processing continues at
19 decision block 422.

20 At decision block 422, a determination is made whether one of the edges
21 (e.g., "false" edge) for statement N is relevant. This determination is based on the
22 concrete state of symbolic state S. For example, if the symbolic state S contained
23 "Flag = True" and the condition for the branch statement was "if Flag", the "T"
24
25

1 edge of statement N would be relevant, but the "F" edge of statement N would not
2 be relevant. When the edge is relevant, processing continues at block 424.

3 At block 424, the concrete state of the copied symbolic state S' is updated
4 to reflect the knowledge that the branch condition is false. Because the process is
5 proceeding down the false path of the branch statement, this knowledge is added
6 to symbolic state S'. Processing continues at block 426.

7 The add heuristic at block 426 is similar to processing described above for
8 block 418. However, the copy of the symbolic state S' is added to the outgoing
9 "False" edge from statement N. The add process is described in detail below in
10 conjunction with FIGURE 5. Processing continues at decision block 428, as does
11 the processing if decision block 422 concludes that the "False" edge is not
12 relevant.

13 At decision block 428, a determination is made whether another edge (e.g.,
14 "True" edge) for statement N is relevant. Again, this determination is based on the
15 concrete state of the current copy of symbolic state S (symbolic state S"). If the
16 edge is relevant, processing continues at block 430.

17 At block 430, the concrete state of the copied symbolic state S" is updated
18 to reflect the knowledge that the branch condition is true. This knowledge is
19 added to symbolic state S" because the process is proceeding down the true path
20 of the branch statement. Processing continues at block 432.

21 The add heuristic performed at block 432 is similar to processing described
22 above for blocks 418 and 426. However, the copy of the symbolic state S" is
23 added to the outgoing edge (e.g., "true" edge) from statement N. Again, the add
24
25

1 process is described in detail below in conjunction with FIGURE 5. Processing
2 then loops back to decision block **402** for another symbolic state, if available.

3 While FIGURE 4 illustrates a process **400** in which the decisional
4 statement is a branch statement, other decisional statements may also be used. If
5 these other decisional statements have more than two paths, additional copies of
6 the symbolic store (block **420**) are created and additional blocks similar to blocks
7 **422 -426** are added for each additional path. Each outgoing edge from a multi-
8 way branch updates the symbolic state according to the condition under which
9 program execution follows that branch. The add heuristic is now described in
10 detail.

11 FIGURE 5 is a logical flow diagram illustrating a process for adding
12 symbolic states to the symbolic store. The process enters at starting block **501**,
13 after a symbolic state has been processed. Processing continues at decision block
14 **502**.

15 At decision block **502**, a determination is made whether another symbolic
16 state exists in the symbolic store for this particular edge. As mentioned above, the
17 symbolic store may contain several symbolic states. Because branch statements
18 sometimes cause two differing symbolic states to be stored in the symbolic store,
19 one edge may have several symbolic states associated with it. The present path-
20 sensitive value flow analysis attempts to merge these symbolic states whenever
21 possible. Therefore, if any symbolic state exists in the symbolic store for the edge
22 associated with statement N, processing continues at block **508**. If no symbolic
23 state exists in the symbolic store for the edge associated with statement N,
24
25

1 symbolic state S is added to the symbolic store and to the worklist, at blocks **504**
2 and **506**, respectively.

3 At block **508**, another symbolic state T for this edge is retrieved from the
4 symbolic store. At decision block **510**, a determination is made whether the value
5 alias information for symbolic states S and T are the same. If the value alias
6 information for the states are different, this indicates that the value being analyzed
7 is different in the two symbolic states. Therefore, the present path-sensitive value
8 flow analysis does not merge the symbolic states. In this case, processing
9 continues to decision block **512**.

10 At decision block **512**, a determination is made whether there is another
11 symbolic state in the symbolic store for this edge. If there is, the process loops
12 back to block **508** and proceeds as described above. However, if there is not
13 another symbolic state for this edge, processing continues at block **504** where the
14 symbolic state S is added to the symbolic store for this edge. The symbolic state S
15 is added because it represents a symbolic state with new value alias information.
16 This will occur whenever the "NO" branch from block **512** is reached. By
17 reaching the "NO" branch, the process determines that the symbolic store does not
18 currently contain any symbolic state with the same value alias information as that
19 of symbolic state S. Thus, in accordance with the present method, the symbolic
20 store will contain, at the most, one symbolic state for each value alias information
21 possible at each edge. Whenever there are two symbolic states with the same
22 value alias information at one edge, the present path-sensitive value flow analysis
23 merges the two symbolic states into a merged symbolic state. The process then
24 proceeds to block **506** where symbolic state S is added to the worklist. By adding
25

1 the symbolic state to the worklist, the process ensures that symbolic state S will be
2 carried forward in processing the remaining paths in the abstract program.

3 Referring back to decision block 510, if the determination concludes that
4 the value alias information is the same, processing continues to block 514. At
5 block 514, the contents of the concrete state in symbolic state S and symbolic state
6 T are merged. In one embodiment, the merge is performed by deleting any
7 information in the concrete state that is different between symbolic states S and T.
8 Symbolic state S is updated with this merged concrete state. In addition, at block
9 516, symbolic state T in the symbolic store is replaced with the newly merged
10 symbolic state S for this edge, thereby reducing the number of symbolic states in
11 the symbolic store.

12 This merging of information and replacement of symbolic states in the
13 symbolic store results in some information being lost. Thus, the precision of the
14 path-sensitive value flow analysis process is decreased. However, the inventors of
15 the present path-sensitive value flow analysis have recognized that as long as the
16 relevant branching behavior is maintained, the accuracy of the present analysis is
17 not appreciably reduced in comparison to the traditional very precise path-
18 sensitive value flow analysis. This heuristic avoids exponential explosion of the
19 search space while still capturing the relevant branching behavior. Therefore,
20 software programs with large code bases may be analyzed with the present path-
21 sensitive value flow analysis. Processing continues to block 506.

22 As described above, at block 506, the symbolic state S is added to the
23 worklist. This "merged" symbolic state then becomes one of the symbolic states
24 in the worklist and is used in further processing. One will note that the effect of
25

1 the merging at block 514 is that for any edge, there will be only one symbolic state
2 in the symbolic store per value alias sets. Therefore, the number of symbolic states
3 in the symbolic store will not grow exponentially due to branches in the program.
4 Processing then continues to return block and is complete.

5 FIGURE 6 is a Table 600 illustrating transform functions for updating a
6 Must set of the value alias information for use in block 413 of FIGURE 4. For
7 those familiar with dataflow analysis, Table 600 illustrates the Generate/Kill sets
8 associated with the present process. Thus, Table 600 includes three columns; a
9 first column (denoted with “v”) identifies types of statements that the value flow
10 analysis tool will encounter during processing of the value alias information. A
11 second column (denoted with “Remove”) represents the transfer function that is
12 applied to the Must set in order to remove value aliases that are no longer value
13 aliases for the analyzed value after processing the statement encountered in the
14 first column. A third column (denoted with “Generate”) represents the transfer
15 function that is applied to the Must set in order to add value aliases that now store
16 the value being analyzed.

17 Each row (e.g., rows 602-616) in Table 600 identifies the Remove transfer
18 function (second column) and the Generate transfer function (third column) that
19 are both associated with the type of statement (first column). Row 602 represents
20 a special statement, “x = Create()”. This statement identifies the starting position
21 for tracking a specific variable (e.g., “x”). When “x=Create()” is encountered, the
22 value alias information is the empty set. Thus, there are not any value aliases to
23 remove, so the remove transfer function is the empty set. The generate transfer
24 function adds the specific variable (i.e., “x”).
25

Before explaining Table 600 in further detail, the symbols used within Table 600 will be explained. Two brackets “{}” represents the empty set. An asterisk “*” represents a wildcard. An arrow “→” represents a pointer dereference. Thus, “ $x \rightarrow *$ ” represents all the fields that the pointer x may dereference. The symbol “ $y \in must$ ” denotes that “ y ” is a member of the Must set. In contrast, the symbol “ $y \notin must$ ” denotes that “ y ” is not a member of the Must set. The standard union symbol “ \cup ” denotes that the actions on both the right side and the left side of the union statement are performed. For example, “ $\{x \rightarrow *\} \cup \{x\}$ ” represents that all the fields that the pointer x may dereference and the variable x should be considered. A conditional is denoted with “|” symbol. The statement to the right of the “|” symbol is the conditional, and the statement to the left of the “|” symbol is the action. Thus, “ $x \mid y \in must$ ” states that if the variable “ y ” is a member of the Must set before the statement is encountered, the variable “ x ” is added or removed from the Must set depending on whether the transfer function is the Remove or Generate function.

The function “*LocAlias*($x \rightarrow f, x$)” checks whether the two passed arguments (i.e., $x \rightarrow f$ and x) point to the same memory location. The function will return a TRUE if the arguments both refer to the same memory location. In general, the *LocAlias* () function is over-inclusive and is commonly referred to as a scalable memory alias analysis. Memory alias analysis is well known in the art. Additional information on one technique for memory alias analysis may be obtained from “*Unification-based pointer analysis with directional assignments*”, in Proceedings of the ACM SIG-PLAN 2000 Conference on Programming Language Design and Implementation, 2000, by Manuvir Das. The function

1 “Mods(v)” identifies all the memory cells (locations) that are updated by the
2 statement v . The function “Refs(v)” identifies all the memory cells (locations) that
3 were looked up when executing the statement v . Both of these functions, Mods(v)
4 and Refs(v), are well known to those skilled in that art and are not discussed here
5 at length.

6 As mentioned earlier, the first column identifies different types of
7 statements that the value flow analysis tool will encounter during processing of the
8 abstract program. Row 604 is a statement that assigns the value of one variable to
9 another variable (e.g., $x=y$). Row 606 is a statement that assigns an address of one
10 variable to another variable (e.g., $x=\&y$). Row 608 is a statement that allocates
11 memory and assigns the result to a variable (e.g., $x=\text{allocate}()$). Row 610 is a
12 statement that assigns a value from a field dereferenced from one variable to
13 another variable (e.g., $x=y\rightarrow f$). Row 612 is a statement that assigns an address of
14 a field pointed to by one variable to another variable (e.g., $x=\&(y\rightarrow f)$). Row 614 is
15 a statement that assigns a value of a variable to a field dereferenced by another
16 variable (e.g., $x\rightarrow f=y$). Row 616 represents the statements, other than the
17 statements described in rows 602-614.

18 The statements will now be described in order to explain Table 6. Row 604
19 is associated with a scalar to scalar assignment statement (e.g., “ $x=y$ ”). For the
20 exemplary statement, the value of the variable “ y ” is copied into the variable “ x ”.
21 The generate function is a union of two sets of information, the first set states that
22 if the variable “ y ” is currently a member of the Must set, add the variable “ x ” to
23 the Must set. This is done because if the variable “ y ” already contained the value
24 being analyzed, the variable “ x ” now contains the same value. The other set states
25

1 that if a field (e.g., "f") dereferenced by pointer "y" is a member of the Must set,
2 add the field (e.g., "f") that can be dereferenced by pointer "x" to the Must set.
3 This is done because if "y" was a pointer, after assigning "y" to "x", "x" will point
4 to the same memory locations as the "y" pointer.

5 The remove transfer function is also the union of two sets of information.
6 The first set states that all the fields that can be dereferenced with x need to be
7 removed from the Must set. The second set has a conditional with a union. The
8 conditional is based on whether the variable y was a member of the Must set when
9 the statement was encountered. If the variable y was a member of the Must set, no
10 value aliases are removed. However, if the variable y was not a member of the
11 Must set, the variable to the left of the assignment (i.e., "x") is removed. In
12 addition, if "z→f" and "x" reference the same memory location, the value alias
13 "z→f" is removed.

14 This occurs because if "x" was in the Must set before the statement was
15 encountered, once the value of the variable "y" is assigned to the variable "x", the
16 variable "x" only needs to be tracked if the variable "y" was in the Must set.
17 Otherwise, the variable "x" is no longer of interest for analyzing the value flow.
18 Likewise, any value alias of "x" should be removed.

19 For row 606, the generate transfer function adds the value alias
20 "x→NULL" if the variable y was in the Must set before encountering the
21 statement. The remove transfer function removes any field that x previously
22 dereferenced (e.g., x→*), the variable itself (e.g., x), and any memory location
23 (e.g., z→f) that pointed to the same memory location as the variable that was
24 assigned (e.g., x).
25

1 For row 608, the generate transfer function does not add any additional
2 value aliases. The remove transfer function removes any field that x previously
3 dereferenced (e.g., $x \rightarrow *$), the variable itself (e.g., x), and any memory location
4 (e.g., $z \rightarrow f$) that pointed to the same memory location as the variable that was
5 assigned (e.g., x).

6 For row 610, the generate transfer function adds the value alias "x" if the
7 field referenced by variable y (e.g., $y \rightarrow f$) was in the Must set before encountering
8 the statement. The remove transfer function removes any field that x previously
9 dereferenced (e.g., $x \rightarrow *$). In addition, the remove transfer function removes the
10 variable itself (e.g., x) and any memory location (e.g., $z \rightarrow g$) that pointed to the
11 same memory location as the variable that was assigned (e.g., x), if the assigned
12 field (e.g., $y \rightarrow f$) was not a member of the Must set before the statement was
13 encountered.

14 For row 612, the generate transfer function adds the value alias
15 " $x \rightarrow \text{NULL}$ " if the field pointed to by variable y (e.g., $y \rightarrow f$) was in the Must set
16 before encountering the statement. The remove transfer function removes any
17 field that x previously dereferenced (e.g., $x \rightarrow *$), the variable itself (e.g., x), and
18 any memory location (e.g., $z \rightarrow g$) that pointed to the same memory location as the
19 variable that was assigned (e.g., x).

20 For row 614, the generate transfer function adds the left-hand variable (e.g.,
21 $x \rightarrow f$) if the right-hand variable (e.g., y) was in the Must set before encountering
22 the statement. The remove transfer function removes any fields pointed to by a
23 variable (e.g., $z \rightarrow *$) if the variable (e.g., z) previously pointed to the same memory
24 location as the left-hand variable (e.g., $x \rightarrow f$). In addition, the remove transfer
25

1 function removes the left-hand variable (e.g., $x \rightarrow f$) if the right-hand variable (e.g.,
2 y) was not in the Must set before encountering the statement. The remove transfer
3 function also removes any memory locations that pointed to the same memory as
4 the left-hand variable (e.g., $\{z \rightarrow g \mid \text{LocAlias}(x \rightarrow f, z \rightarrow g)\}$ and $\{z \mid$
5 $\text{LocAlias}(x \rightarrow f, z)\}$) if the right-hand variable (e.g., y) was not in the Must set
6 before the statement was encountered.

7 For row 616, the generate transfer function does not add any alias
8 information. The remove transfer function removes any variable (e.g., x) that
9 pointed to the same memory location as a memory cell that was updated by the
10 statement (e.g., $\text{LocAlias}(x, \text{Mods}(v))$). In addition, the remove transfer function
11 removes any field (e.g., $x \rightarrow f$) if the memory cell updated by the statement
12 (determined by $\text{Mods}(v)$) pointed to the same memory location as the field ($x \rightarrow f$)
13 or the pointer (x) (e.g., $\{x \rightarrow f \mid \text{LocAlias}(x, \text{Mods}(v)) \vee \text{LocAlias}(x \rightarrow f, \text{Mods}(v))$
14 $\}$).

15 In general, the inventors of the present path-sensitive value flow analysis
16 discovered that even though these memory alias analyses tend to be inaccurate,
17 their results may be used to rule out a large number of irrelevant assignments
18 through pointers. In addition, instead of introducing all possible memory aliases
19 into the value alias information, the present analysis uses a placeholder which can
20 then be expanded using memory alias analysis on demand. Thus, by implicitly
21 representing the value alias sets, precise value flow analysis and imprecise
22 memory alias analysis may be combined to create a feasible path-sensitive value
23 flow analysis.
24
25

FIGURE 7 is a table illustrating transform functions for updating the May set of the value alias information for use in block 413 of FIGURE 4. Similar to Table 6, Table 700 includes three columns; a first column (denoted with “v”) identifies types of statements that the value flow analysis tool will encounter during processing of the abstract state. A second column (denoted with “Remove”) represents the transfer function that is applied to the May set in order to remove value aliases that are no longer value aliases for the analyzed value after processing the statement encountered in the first column. A third column (denoted with “Generate”) represents the transfer function that is applied to the May set in order to add value aliases that may now store the value that is being analyzed.

The different types of statement listed in the first column of Table 700 correspond to the types of statements listed in the first column of Table 600. Therefore, the following will describe the generate and remove transfer functions for each of the different types of statements.

For row 704, the generate transfer function adds the left-hand variable (e.g., x) if the right-hand variable (e.g., y) pointed to the same memory location as any alias in the May set (e.g., LocAlias(y, may)). The remove transfer function removes the left-hand variable (e.g., x).

For rows 706, 708, and 712, the generate transfer function does not add an alias to the May set. The remove transfer function removes the left-hand variable (e.g., x).

For row 710, the generate transfer function adds the left-hand variable (e.g., x) if the right-hand variable (e.g., $y \rightarrow f$) pointed to the same memory location as

1 any alias in the May set (e.g., `LocAlias(y→f, may)`). The remove transfer
2 function removes the left-hand variable (e.g., `x`).

3 For row 714, the generate transfer function adds the left-hand variable (e.g.,
4 `x→f`) if the right-hand variable (e.g., `y`) pointed to the same memory location as
5 any alias in the May set (e.g., `LocAlias(y, may)`). The remove transfer function
6 does not remove any value aliases.

7 For row 716, the generate transfer function adds an expression (e.g., `e`) if
8 the expression (e.g., `e`) was updated by the statement (e.g., `e ∈ Mods(v)`) or if one
9 of the memory locations looked up when executing the statement pointed to a
10 memory location in the May set (e.g., `LocAlias(Refs(v), may)`).

11 Upon reviewing the transfer functions for the May set, one will note that
12 the transfer functions do not add all the memory aliases to the May set, rather, the
13 May set implicitly represents its memory aliases. Therefore, the Remove transfer
14 functions do not remove expressions from the May set due to an assignment,
15 except for variables.

16 Now, the flow of process 400 and process 500 will be described in
17 conjunction with the control flow graphs shown in FIGURES 8-10 and the transfer
18 functions shown in FIGURES 6-7. Briefly, the control flow graphs shown in
19 FIGURES 8-10 are identical and visually represent the abstract program created
20 from the source code shown in Table 1 below.

22 Table 1:

```
23 void entryPoint(int *o) {  
24     if (b)  
25         p=o;
```

```

1      else
2          p = getPointer();
3
4      if (c)
5          x=1;
6      else
7          x=0;
8
9      if (b)
10         checkPointer(p);
11
12     data = *p;
13
14     }.

```

End of Table 1.

Control flow graph **800** includes fourteen nodes N0-N13. Each node is associated with one of the statements in the source code shown in Table 1. In essence, control flow graph **800** provides a visual representation of the abstract program created in block **302** of FIGURE 3. As shown in Table 1, the program includes one function named `entryPoint()` that is passed an argument "z". Thus, the following discussion of the example source code describes the process for tracking the concrete and value alias information with respect to the flow of the value initially stored in variable "z". The join points (e.g., nodes N4, N8, and N11) associated with each branch statement (e.g., nodes N1, N5, and N9) are explicitly identified in the CFG.

1 The process enters the `entryPoint()` program (begin block **401**) represented
2 by node **N0** in **FIGURE 8**. Prior to entering `entryPoint()` or upon entering
3 `entryPoint()`, the assignment statement "`create()`" is specified for the variable "`z`".
4 Referring to **FIGURES 6 and 7**, rows **602** and **702** for the Must and May sets,
5 respectively, the generate functions for both the Must and May sets are applied.
6 After applying the generate function for the Must set, the Must set contains the
7 variable "`z`". One will note that the generate function for May set also adds the
8 variable "`z`" to the May set. However, because the Must set already includes the
9 variable "`z`", the variable is not included in the May set. At this node, the
10 symbolic store and the worklist both contain one symbolic state (e.g., symbolic
11 state **802**), which includes the variable being analyzed (e.g., "`z`") and an open
12 concrete state indicating that there are no current variables that will limit the
13 execution paths. Thus, the symbolic state **802** is shown having the value alias
14 information of "`z`" in the Must set (denoted by braces without the question mark)
15 and an empty set in the May set (denoted by braces with the question mark) and an
16 open concrete state. The process **400** continues at block **402**.

17 At block **402**, a determination is made whether there is a symbolic state in
18 the worklist. As stated above, there is currently one symbolic state (e.g., symbolic
19 state **802**). Therefore, symbolic state **802** is retrieved from the worklist (block
20 **404**) and removed from the worklist (block **406**). At this point, the worklist does
21 not contain any symbolic states. Processing continues to block **408**.

22 At block **408**, a statement is retrieved that follows the symbolic state **802**
23 (i.e., "`if (b)`" at node **N1** in **FIGURE 8**). Because this statement is a branch
24 (decision block **410**), two copies (not shown) of symbolic state **802** are created
25

1 (block 420). The symbolic state 802 does not rule out proceeding through the
2 "false" path (decision block 422). Therefore, the "false" edge of the "if (b)"
3 statement is relevant for symbolic state 802. The first copy of the symbolic state
4 812 is updated with the predicate of the branch statement (i.e., "b=0" indicated in
5 FIGURE 8 with a line over "b") (block 424). Because there are not any other
6 symbolic states associated with the "false" edge (decision block 502), symbolic
7 state 812 is added to the symbolic store (block 504) and added to the worklist
8 (506).

9 Likewise, the symbolic state 802 does not rule out proceeding through the
10 "true" path (decision block 428). Therefore, the "true" edge of the "if(b)"
11 statement is relevant for symbolic state 802. The second copy of the symbolic
12 state 811 is updated with the predicate of the branch statement (i.e., "b=1") (block
13 430). Again, because there is not another symbolic state associated with "true"
14 edge (decision block 502), symbolic state 811 is added to the symbolic store
15 (block 504) and added to the worklist (506). At this time, the symbolic store
16 contains symbolic states 802, 811, and 812. The worklist contains symbolic states
17 811 and 812.

18 Returning to decision block 402, either one of these symbolic states from
19 the worklist may be retrieved. Assume that symbolic state 811 is retrieved (block
20 404) and removed (block 406) from the worklist. The "p=z" statement at node N2
21 is retrieved (block 408). The "p=z" statement is not a branch (decision block 410),
22 but does affect the symbolic state of 811. Therefore, symbolic state 811 is updated
23 (block 413). Updating symbolic state 811 involves applying the Must and May
24 transfer functions applicable to the "p=z" statement. Thus, the transfer functions
25

1 in rows 604 and 704 are applied. Applying the transfer functions for the Must set
2 (row 604) adds the variable "p" to the Must set because the variable "z" was in the
3 Must set prior to the statement. Applying the transfer functions for the May set
4 (row 704) adds no additional value alias because the May set was empty before the
5 statement. Thus, after applying the transfer functions for the Must and May sets,
6 symbolic state 821 results (block 414). Because there is not any other symbolic
7 state in the symbolic store for outgoing edge of node N2 (decision block 502),
8 symbolic state 821 is added to the symbolic store (block 504) and added to the
9 worklist (block 506). At this time, the symbolic store contains symbolic states
10 802, 811, 812, and 821. The worklist contains symbolic states 812 and 821.

11 Returning to decision block 402, either one of these symbolic states may be
12 retrieved. Assume that symbolic state 812 is retrieved (block 404) and removed
13 (block 406). The "p=getPointer()" statement at node N3 is retrieved (block 408).
14 The "p=getPointer()" statement is not a branch (decision block 410), and does not
15 affect the symbolic state of 812. Therefore, symbolic state 811 is copied to
16 symbolic state 823 (block 416). Again, because there is not a symbolic state in the
17 symbolic store for the edge (block 502), Symbolic state 823 is added to the
18 symbolic store (block 504) and added to the worklist (block 506). At this time, the
19 symbolic store contains symbolic states 802, 811, 812, 821, and 823. The worklist
20 contains symbolic states 821 and 823.

21 Returning to decision block 402, either one of these symbolic states may be
22 retrieved from the worklist. Assume that symbolic state 821 is retrieved (block
23 404) and removed (block 406). The join statement at node N4 is retrieved (block
24 408). The join statement is not a branch (decision block 410), and does not affect
25

1 the symbolic state **821**. Therefore, symbolic state **821** is copied to symbolic state
2 **831** (block **416**). Because there is not a symbolic state in the symbolic store for
3 the edge (block **502**), symbolic state **831** is added to the symbolic store **841** for
4 this edge (block **504**) and added to the worklist (block **506**). At this time, the
5 symbolic store contains symbolic states **802**, **811**, **812**, **821**, **823**, and **831**. The
6 worklist contains symbolic states **823** and **831**.

7 Returning to decision block **402**, there is another symbolic state (symbolic
8 state **823**) in the worklist. Symbolic state **823** is retrieved (block **408**) and
9 removed (block **406**). The join statement at node N4 is retrieved (block **408**).
10 Because the join statement is not a branch statement (decision block **410**) and does
11 not affect the symbolic state **823**, processing proceeds to decision block **502** in
12 FIGURE 5. However, this time there is a symbolic state in the symbolic store for
13 the join state at node N4 (i.e., symbolic state **831**). Thus, symbolic state **831** is
14 retrieved (block **508**). Because the value alias information is different (decision
15 block **510**) and there is not another symbolic state in the symbolic store for this
16 edge (decision block **512**), symbolic state **823** is added to the symbolic store
17 (block **504**) and to the worklist (block **506**) as symbolic state **841**. Thus, node
18 N4 merges the incoming facts and keeps the execution states of b and not b
19 separate because the associated value alias sets are different. Thus, at this point,
20 the symbolic store contains symbolic states **802**, **811**, **812**, **821**, **823**, and **841**. The
21 worklist contains symbolic states **841**. It is important to note that traditional data
22 flow analysis would have merged all the execution states into one, thereby losing
23 precision compared with the present process.

1 One will note that the processing of each symbolic state at each node may
2 proceed independently. The add heuristic will properly update the symbolic store
3 at the appropriate time. Therefore, in this embodiment, the process does not have
4 to completely finish both paths of a branch statement before continuing with the
5 join statement and before processing other statements.

6 The symbolic state **841** that is in the worklist is then processed through
7 process **400** and **500** and the newly created symbolic states are added to the
8 worklist. The newly created symbolic states are also processed through process
9 **400** and **500**. FIGURES 9 - 10 illustrate the symbolic states that occur at each
10 node. Without being unnecessarily duplicative, the following discussion only
11 describes the determination of the Must and May set for each statement and does
12 not walk through the entire processing for merging the symbolic states at join
13 statements. One skilled in the art can easily expand on the foregoing example
14 illustrated in FIGURE 8 and described above to create the symbolic states shown
15 in FIGURES 9-10.

16 At node N5, given that either b is "0" or "1", the process can determine that
17 one of the branches is irrelevant. Therefore, the execution state is duplicated and
18 one copy traverses through each branch. The statement "if(c)" affects only the
19 concrete state of the symbolic state **841** by adding "C=1" or just "C" to the
20 concrete state (see symbolic state **941**). Likewise, the statement "if(c)" affects the
21 concrete state of the symbolic state **841** by adding "c=0" to the concrete state (see
22 symbolic state **942**). The Must set and the May set are not changed after applying
23 the transfer functions identified in rows **616** and **716**, respectively. Thus, both of
24
25

1 the symbolic states, symbolic states **941** and **942** are propagated to the true and
2 false successors of node N5, respectively.

3 Again, at node N6 and N7, only the concrete state is affected by adding the
4 value of “x” to the concrete state. Because these nodes do not affect the value
5 alias information, the symbolic states for these nodes were omitted.

6 At node N8, the symbolic states **941** and **942** are merged based on the value
7 alias set. The resulting symbolic state **951** states that either the Must set is {z,p}
8 and the May set is empty when b is true, or the Must set is {z} and the may set is
9 empty when b is false. Thus, the process dropped the value of “c” from the
10 execution state because it was not correlated with the value alias set for the value
11 of interest. This is in contrast with traditional path-sensitive value flow analysis
12 which would have continued tracking c accurately and double the number of
13 execution states that were analyzed downstream of node N8.

14 At node N9, FIGURE 10, the process determines that only one leg of the
15 branch is feasible: true for state b and false for not b. Thus, the symbolic states
16 **961** and **962** are propagated to the true and false successors of node N9,
17 respectively.

18 At node N10, FIGURE 10, symbolic state **961** indicates that “p” holds the
19 tracked value. Therefore, the call to CheckPointer() performs the check on the
20 tracked value, and the resulting symbolic state **971** is marked as checked
21 (indicated with a superscript “c”). Again, this is in contrast with traditional data
22 flow analysis which would have only indicated the variable “p” *may* hold the
23 tracked value. Therefore, using traditional data flow analysis, symbolic state **971**
24 would not have been able to be marked as checked.
25

1 At node N11, FIGURE 11, symbolic states 962 and 971 are merged to
2 create symbolic state 981. At node N12, the variable “p” is dereferenced (e.g.,
3 data = *p). Two states may reach this statement. Symbolic state 971 states that
4 the variable “p” holds a pointer that has been checked. Therefore, the dereference
5 at node N12 is valid. The other symbolic state 962 states that the variable “p”
6 does not hold a user-mode pointer. Therefore, the dereference is valid. Using
7 traditional, data flow analysis, the dereference would have been invalid and a false
8 error would have been reported.

9 As one skilled in the art will appreciate, the present path-sensitive value
10 flow analysis also operates with nested decisional statements or loop constructs.
11 The inner nests, along with their symbolic stores, are processed before proceeding
12 with any outer nests. In addition, for the above discussion, the intraprocedural
13 aspect of the present analysis is described in detail. The present path-sensitive
14 value flow analysis also operates in an interprocedural manner. For programs with
15 multiple procedures and calls between procedures, the analysis produces a
16 combined CFG by combining the CFGs of individual procedures, in a manner that
17 is well known in the art. For example, edges are added that connect a call site
18 with the entry point of the called function, and that connect the exit point of the
19 called function with a corresponding return node for the call site. Processing is as
20 in the intraprocedural case, except that symbolic states are not merged at the exit
21 points of functions. Instead, the information at exit points is stored as a mapping
22 from a symbolic state at the entry point to the resulting symbolic state at the exit
23 point.

1 Thus, as described above, the path-sensitive value flow analysis tracks the
2 flow of values through a program and eliminates value flow information from
3 infeasible execution paths. Thus, the analysis scales easily to a million lines of
4 code and is path-sensitive. As described above, the analysis achieves these results
5 by tracking the flow of one value at a time, merging value flow information from
6 different execution paths if the value flow information is the same, and applying
7 complex aliasing information in a manner so that not all memory aliases need to
8 be added.

9 Although details of specific implementations and embodiments are
10 described above, such details are intended to satisfy statutory disclosure
11 obligations rather than to limit the scope of the following claims. Thus, the
12 invention as defined by the claims is not limited to the specific features described
13 above. Rather, the invention is claimed in any of its forms or modifications that
14 fall within the proper scope of the appended claims, appropriately interpreted in
15 accordance with the doctrine of equivalents.